Fault-Prone Java Method Analysis Focusing on Pair of Local Variables with Confusing Names

Keiichiro Tashima*, Hirohisa Aman†, Sousuke Amasaki‡, Tomoyuki Yokogawa‡ and Minoru Kawahara†
*Department of Computer Science, Faculty of Engineering, Ehime University
Matsuyama, Ehime, 790–8577 Japan
†Center for Information Technology, Ehime University
Matsuyama, Ehime, 790–8577 Japan
‡Faculty of Computer Science and Systems Engineering, Okayama Prefectural University
Soja, Okayama, 719–1197 Japan

***Abstract*—Giving a name to a local variable is usually a programmer's discretion. Since it depends on the programmer's preference and experience, there is a lot of individual variation which may cause a variability in the code quality such as the readability. While there have been studies on the naming of local variables in the past, a relationship of names among local variables within a method (function) has not been well-discussed. This paper focuses on a pair of local variables with similar, confusing names, e.g., "lineIndex" vs. "lineIndent." Since such local variables are confusable with each other, the presence of such a confusing pair may be related to the fault-proneness of the method. An empirical analysis for five major open source Java projects is conducted, and the following results are reported: (1) a method having a confusing variable pair is about 1.1 – 2.6 times more fault-prone than a method having only dissimilar (non-confusing) pairs; (2) the proposed metric of how confusing the local variables are is equivalent to or better than the conventional cyclomatic complexity in predicting fault-prone methods.**

I. Introduction

A proper quality management of source code is essential to a successful software development and maintenance. Since a coding activity is an intellectual task by a programmer, i.e., a human being, an individual variation in the code quality would be inevitable. One of the most diverse artifacts in source code is a local variable in a method (function). Since the requirements specification and the design documents usually do not specify the declaration and use of local variables, programmers can freely decide them during their coding activity. Different programmers may give different names to a local variable for the same purpose. Such an individual difference causes a variation of the code quality such as the readability. In general, names (identifiers) are important information source to comprehend the code [1]. In other words, the quality of name plays a significant role to lead the code readers to a proper understanding of the code; meaningless or improper names degrade the source code [2], and may cause a fault creation or an overlooking of fault.

There have been studies focusing on variable names (or identifiers in the more general sense) in the past [3], [4], [5], [6], [7], [8], [9]. For example, they analyzed the length and the composition of local variable's name, and discussed the relationships with the code comprehension or the code quality. While these previous work presented actionable results, their main focuses were on the "individual names" but not the "relationships between names." As a novel point of view, we focus on the similarity between local variables' names. If there is a pair of local variables with highly-similar names, they would be confusable with each other: for example, "lineIndex" vs. "lineIndent." The presence of such a confusing pair may have an impact on the code quality. In this paper, we propose to quantify the degree to which local variables' names are confusing, and perform an empirical study to examine its impact on the fault-proneness of methods.

The remainder of this paper is organized as follows. Section II briefly describes the related work focusing on local variables' names, and presents our research motivation. Section III introduces the key notion of "confusing names" of local variables and defines a metric to measure the degree to which two names are confusing. Section IV reports and discusses the results of our empirical study. Finally, Section V presents the conclusion of this paper and our future work.

II. Related Work

Lawrie et al. [3] focused on the composition of variable's name, and conducted a comparative survey on the program comprehension with using three types of names: a fully-spelled word, an abbreviated word and a single character, e.g., "index," "idx" and "i," respectively. They showed that the understandability of a name decreases in the order of a fully-spelled one, an abbreviated one and a single-character one, but there is no significant difference between the fully-spelled name and its abbreviated one in the comprehension by programmers. A similar trend is supported by the large-scale experiment which was conducted and reported by Scanniello et al. [4]. Kawamoto and Mizuno [5] focused on the length of identifier and reported that a class having a long identifier tends to be fault-prone. Aman et al. [7] also reported that a method having a local variable with a long name is change-prone and cannot survive unscathed. Although a long name of a variable can describe its role more accurately, a long name may decay the readability of code [6]. Kernighan and Pike [10] said that it is overdone to use a long and descriptive name for a variable if its scope is narrow.

Binkley et al. [8] compared the impact of difference in the naming style—the camel case and the snake case, e.g., "maxValue" vs. "max_value"—on the program comprehension. As the result, they reported that the camel case is better for programming beginners in terms of the comprehension while there is no significant difference for experts. Bulter et al. [9] proposed 12 naming rules and showed that programs whose identifiers are against their rules are fault-prone.

As the above studies reported, a name of variable (identifier) is a noteworthy feature in the program comprehension and quality management. To the best of our knowledge, however, the previous work focused on the "individual" name; a "relationship between names" has not been well-discussed in the past. When a method has two or more local variables, the similarity among them would be yet another interesting point of view. This is our research motivation in this paper.

III. Confusing Names of Local Variables

As we mentioned above, we focus on a relationship between local variables' names in this paper. Notice that our analysis is limited to local variables although there are also other types of identifiers including class names, class field (class or instance variable) names and method names. Since classes, fields and methods might be specified in the design phase, programmers would not be able to decide their names freely. On the other hand, names of local variables are almost always given by programmers at their own discretion during the programming phase. In order to support the programming activity from the perspective of naming, we decided to analyze local variables in this paper; a further analysis involving other types of identifiers is our future work.

A. Pair of Highly-Similar (Confusing) Names

In general, a local variable should have an proper name to express its role. However, when two or more local variables are declared in a method, we may need to take care of their similarities as well. If there is a pair of highly-similar names, one variable is confusable with another variable. Such a confusing pair might cause a misuse of variable which is a fault creation, or an overlooking of fault.

Figure 1 presents an example of code fragment. In the figure, the program logic is simple and each local variable has a well-described name. However, it does not look a "simple" code; there is a pair of local variables with long and similar (confusing) names—"distanceBetweenAbscissae" and "distanceBetweenOrdinates." We might make a mistake when we copy and paste some statements or when we use the code completion function on an integrated development environment such as Eclipse.

```
.....
distanceBetweenAbscissae = firstAbscissa – secondAbscissa;
distanceBetweenOrdinates = firstOrdinate – secondOrdinate;
if ( distanceBetweenAbscissae ∗ distanceBetweenAbscissae
        > distanceBetweenOrdinates ∗ distanceBetweenOrdinates ){
.....
```

Fig. 1. A code fragment having a pair of local variables with confusing names.

Now we define highly-similar names as "confusing" names. Hereinafter, we will call a pair of local variables with confusing names as a pair of "confusing local variables" or simply "confusing variables."

B. Levenshtein Distance

To evaluate the degree to which two names are confusing, we will leverage the Levenshtein distance between them in this paper.

Basically, the notion of confusing names has two different points of view: the string similarity and the semantic similarity. For example, "levelOfStrength" and "levelOfStrangeness" are similar strings, but their meanings are not close. We also tried evaluating the semantic similarity by using the Word2Vec [11]. However, we faced a difficulty to appropriately handle abbreviated words in the semantic analysis. For example, we have to appropriately expand "idx" to "index" so that the Word2Vec can process it. Hence, we decided to start tackling the string similarity in this paper; we would like to perform a further analysis focusing on the semantic similarity as our important future work.

The Levenshtein distance between two strings is the minimum number of the character editing operations which are required to change one string to another string, where a character editing operation is one of the following operations: (1) a single character deletion, (2) a single character insertion, and (3) a single character substitution. For example, the Levenshtein distance between "first" and "last" is 3 because "first" $\xrightarrow{\text{f}\to\text{l}}$ "lirst" $\xrightarrow{\text{i}\to\text{a}}$ "larst" $\xrightarrow{\text{delete r}}$ "last."

C. Normalized Levenshtein Distance

Although the Levenshtein distance can be a measure of string similarity, it has an unsuitable feature to evaluate the degree to which two names are confusing. Let us take the following two pairs for example:

1) "get" and "set" ,
2) "dXAxisTitleThickness" and "dYAxisTitleThickness" .

Both pairs have the same Levenshtein distance, 1. However, pair 2) seems to be more confusing than pair 1). Since such an abnormality is from the difference of their string lengths, we propose the following normalized Levenshtein distance (NLD) between two strings (local variables' names) s_1 and s_2 as our measure of how confusing these two variables are:

$$\mathrm{NLD}(s_1, s_2) = \frac{\mathrm{LD}(s_1, s_2)}{\max\{\ \lambda(s_1), \lambda(s_2)\ \}}, \tag{1}$$

where $\mathrm{LD}(s_1, s_2)$ is the (original) Levenshtein distance, and $\lambda(\cdot)$ signifies the length of the corresponding string.

We get the following evaluations: NLD("get", "set") $= 1/3$ and NLD("dXAxisTitleThickness", "dYAxisTitleThickness") $= 1/20$. That is to say, the latter pair is evaluated as more similar than the former pair.

IV. Empirical Study

To examine the impact of confusing local variables on the fault-proneness of methods, we conduct an empirical study.

A. Dataset

We analyze five major open source projects shown in Table I. In the table, "# files" signifies the number of source files surveyed, excluding test programs and documents[1]. The main reasons why we targeted these projects are as follows. 1) Their source files are written in Java; 2) Their source files are maintained by using the Git; 3) Their fault data are available.

The requirements 1) and 2) are from our data collection tools. The requirement 3) is essential to the fault-prone method analysis. In this study, we leverage the fault data from the tera-PROMISE repository[2].

B. Procedure

We conduct our data collection and analysis in the following procedure.

1) Extract methods and their local variables from Java source files.
   By analyzing Java source files using Eclipse JDT, we extract methods and their local variables from source files. In this study, we consider method parameters to be local variables as well.
   Notice that we exclude the following methods since we cannot compute NLD for variable pairs: (a) a method having no local variable, (b) a method having only one local variable, and (c) an abstract method.
2) Get NLD for each method.
   In each method, compute NLD for each pair of local variables. Since our main focus is on whether the presence of confusing variable pair is related to the fault-proneness of the method or not, we adopt the minimum NLD (the highest similarity) as the measure of the method when there are two or more variable pairs.
3) Divide the method set into subsets by NLD, and compare the fault-proneness among subsets.
   To compare methods according to their NLD, we divide the set of methods into four subsets G_1, G_2, G_3 and G_4, based on the quartile of NLD distribution, where
   - G_1: NLD $\leq$ 25 percentile;
   - G_2: 25 percentile $<$ NLD $\leq$ median;
   - G_3: median $<$ NLD $\leq$ 75 percentile;
   - G_4: 75 percentile $<$ NLD.

   Then, we compute the rate of faulty methods in each subset (fault rate: FR) and compare FR values among G_i (for $i = 1, 2, 3, 4$). In this paper, we use FR as our measure of the fault-proneness of method.
   Notice that the fault data of the surveyed projects from the tera-PROMISE repository is at file-level. To detect faulty methods, we checked which methods are changed through fault fixes.

[1]We omitted source files whose paths contain "test" or "documentation."
[2]http://openscience.us/repo/

TABLE I
Numbers of surveyed files, methods and local variables.

project	# files	# methods	# local variables
Apache Tomcat	1,726	7,051	32,139
BIRT	8,232	34,027	156,786
Eclipse JDT UI	10,452	18,571	87,568
Eclipse Platform UI	4,272	15,937	65,160
SWT	1,731	11,157	64,770
total	26,413	86,743	406,423

C. Results

Figure 2 and Table II present the FRs in G_1 – G_4 for each project. From Fig. 2, we see monotonically decreasing trends of FR from G_1 to G_4 in four out of five projects except for SWT: FR becomes lower as NLD gets larger (from G_1 to G_4). A method category with larger NLD is the set of methods having only pairs of local variables whose names are less similar each other. Thus, a method having only non-confusing local variable pairs is less fault-prone. In other words, a method having a pair of more confusing local variables is more likely to be faulty. FR values change from G_1 (with the most confusing pairs) to G_4 (with the least confusing pairs) as follows.

- Apache Tomcat: FR in G_1 is about 2.6 times higher than that in G_4;
- BIRT: about 1.4 times higher;
- Eclipse JDT UI: about 1.6 times higher;
- Eclipse Platform UI: about 1.5 times higher.

SWT has a different trend: FR in G_2 is higher than that in G_1. But it shows a monotonic decrease from G_2 to G_4 as well, and moreover, FR in G_1 is about 1.1 times higher than that in G_4. Thus, it still seems to be that a method having a confusing local variable pair tends to be more fault-prone.

D. Discussions

In order to examine the relationship between the presence of confusing local variables in a Java method and the fault-proneness of the methods, we divided the method set into four subsets G_1 – G_4 by NLD, and compared their FRs. G_1 is the set of methods having the most confusing local variable pairs,

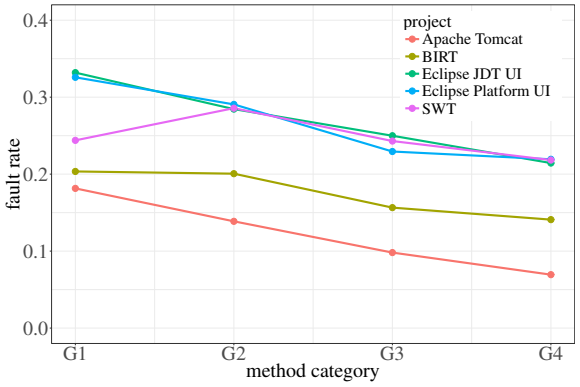

Fig. 2. Fault rates (FRs) in method categories.

TABLE II
FAULT RATES (FRS) IN METHOD CATEGORIES.

project	G_1	G_2	G_3	G_4
Apache Tomcat	0.1815 ($\frac{399}{2198}$)	0.1387 ($\frac{208}{1500}$)	0.0981 ($\frac{162}{1652}$)	0.0694 ($\frac{118}{1701}$)
BIRT	0.2035 ($\frac{1827}{8977}$)	0.2006 ($\frac{1635}{8148}$)	0.1565 ($\frac{1400}{8942}$)	0.1409 ($\frac{1122}{7960}$)
Eclipse JDT UI	0.3319 ($\frac{1825}{5499}$)	0.2845 ($\frac{1208}{4246}$)	0.2499 ($\frac{1151}{4605}$)	0.2144 ($\frac{905}{4221}$)
Eclipse Platform UI	0.3258 ($\frac{1303}{4000}$)	0.2906 ($\frac{1266}{4357}$)	0.2294 ($\frac{831}{3622}$)	0.2193 ($\frac{868}{3958}$)
SWT	0.2439 ($\frac{714}{2928}$)	0.2857 ($\frac{760}{2660}$)	0.2430 ($\frac{761}{3132}$)	0.2183 ($\frac{532}{2437}$)

and the confusing level decreases in the order of G_1 to G_4. As shown in Fig. 2, the FR tends to get lower as the confusing level decreases from G_1 to G_4: four out of five projects except for SWT show monotonic decreases. While SWT presents a different tendency, it also has a monotonically decreasing trend from G_2 to G_4, and the FR in G_1 is still higher than G_4.

By comparing FR values, we see that methods in G_1 are about 1.1 to 2.6 times more fault-prone than ones in G_4. To check the difference in FR values between G_1 and G_4, we additionally performed χ^2 test for each project data. As the results, we confirmed that their differences in all projects are statistically significant at a 5% significance level[3]. Thus, a method having a pair of confusing variables is more likely to be faulty than a method having only non-confusing pairs.

If there is a pair of confusing (highly-similar) local variables in a method, one variable may be confusable with another variable. Such a confusable state may be related to a lack of clearness or readability in the program, and causes a fault creation or an overlooking of fault. However, the appearance of such a confusing variable pair may be related to the size or complexity of the program as well. That is to say, as programs become larger or more complex, programmers tend to use more variables, so the possibility of appearing such a confusing variable pair increases. Thus, we also compared NLD with the conventional size metric (lines of code: LOC) and the code complexity metric (cyclomatic complexity [12]: CC)—we built a fault-prone method prediction model using NLD together with LOC and CC for each project. If NLD is worthless, it cannot contribute to the prediction because LOC and CC work dominantly.

Here we used the random forest as our prediction model [13] since it is one of the most promising models for predicting fault-prone programs [14]. In a random forest, we can evaluate the importance of each variable (metric) by Breiman's method [13]. Table III shows the importance values computed for three metrics. As the results, while NLD does not have the highest importance, it has higher value than CC except for SWT. Although the importance of NLD in SWT is less than that of CC, they seem to be at almost the same level. Therefore, NLD would be one of useful metrics for predicting fault-prone methods, and it is equivalent to or better than CC in predicting fault-prone methods.

[3]For each project, we performed a χ^2 test on the null hypothesis that the FR in G_1 is equal to the FR in G_4.The results are: (Apache Tomcat) $\chi^2 = 103.90$, $df = 1$, $p < 2.2 \times 10^{-16}$; (BIRT) $\chi^2 = 114.41$, $df = 1$, $p < 2.2 \times 10^{-16}$; (Eclipse JDT UI) $\chi^2 = 162.58$, $df = 1$, $p < 2.2 \times 10^{-16}$; (Eclipse Platform UI) $\chi^2 = 113.09$, $df = 1$, $p < 2.2 \times 10^{-16}$; (SWT) $\chi^2 = 4.73$, $df = 1$, $p = 0.02968$.

TABLE III
IMPORTANCE VALUES OF METRICS IN EACH PROJECT'S RANDOM FOREST.

	metric		
project	NLD	LOC	CC
Apache Tomcat	217.7	278.4	171.9
BIRT	722.2	1065.5	535.4
Eclipse JDT UI	546.1	882.3	395.0
Eclipse Platform UI	479.9	786.7	352.3
SWT	411.0	580.9	448.8

Since metric LOC showed the highest importance in the random forest models, we can say again that a larger method tends to be more fault-prone, which has been supported by the previous work (e.g., [15]). This result may be related to the number of local variables as well. That is to say, a larger method is likely to have more local variables within the body, and then the importance of non-confusable naming may get higher. To examine the relationship between NLD and LOC, and to analyze the impact of confusing variable names on the method's fault-proneness in more depth, we plan to perform a further analysis focusing on the details of local variables' properties such as their dependent relationships in the future.

E. Threats to Validity

Since our data analysis is limited to five open source projects, it may be a threat to validity regarding the generality of our results. However, the granularity of our data analysis is at method-level and local variable-level, so we had a lot of samples—$86,743$ methods and $406,423$ local variables as shown in Table I. Moreover, to mitigate the threat, we selected popular projects in which many developers are involved. To assure a higher generality of our findings, we will collect more data and analyze them in the future.

Our data are also limited to Java programs. This limitation is from our data collection tool which we developed to extract local variable's features (names and scopes). Since the notion of local variable is common to modern programming languages, the difference in the programming language may not be a serious threat to validity. To clarify the impact of language difference, we plan to analyze programs written in other languages as our future work.

Our metric NLD focuses only on the highest similarity between two local variables' names within a Java method. Thus, other properties in regard to local variables are not taken into account: the number of local variables, their dependent relationships, etc. Such other properties might also affect the fault-proneness of Java methods. Although we analyzed the effect of NLD together with LOC (method size) anc CC (method complexity) using the random forest, we need to

perform a further analysis using not only these three metrics but also other local-variable-related metrics in the future.

V. Conclusion

We focused on confusing (highly-similar) names of local variables in a Java method, and proposed to quantify the confusing level by the normalized Levenshtein distance (NLD). Then, we conducted an empirical study with five major open source software products, and proved that a Java method having a pair of confusing local variables is about 1.1 to 2.6 times more fault-prone than a method having only non-confusing ones. Moreover, we compared a power of NLD with popular code metrics in terms of the fault-prone method prediction through the random forest, and showed that NLD can probably outperform the cyclomatic complexity. Therefore, focusing on pairs of confusing local variables would be one of beneficial points of view for enhancing the code quality management.

Our future work includes: 1) a further analysis focusing on not only the string similarity but also the meaning closeness between local variables' names; 2) a further investigation on why the appearance of highly-similar variables relates with a fault creation or an overlooking of fault.

Acknowledgment

This work was supported by JSPS KAKENHI #16K00099 and #18K11246. The authors would like to thank the anonymous reviewers for their helpful comments to an earlier version of this paper.

References

[1] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Softw. Quality J.*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[2] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: an analysis of trends," *Empir. Softw. Eng.*, vol. 12, no. 4, pp. 359–388, Aug. 2007.

[3] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proc. 14th Int'l Conf. Program Comprehension*, Jun. 2006, pp. 3–12.

[4] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in c and java source code: Abbreviated vs. full-word identifier names," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 2, pp. 6:1–6:43, Jul. 2017.

[5] K. Kawamoto and O. Mizuno, "Predicting fault-prone modules using the length of identifiers," in *Proc. 4th Int'l Workshop Empir. Softw. Eng. in Practice*, Oct. 2012, pp. 30–34.

[6] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Sc. Comp. Prog.*, vol. 74, no. 7, pp. 430 – 445, May. 2009.

[7] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of change-proneness in methods having local variables with long names and comments," in *Proc. 9th Int'l Symp. Empir. Softw. Eng. & Measurement*, Oct. 2015, pp. 50–53.

[8] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empir. Softw. Eng.*, vol. 18, no. 2, pp. 219–276, Apr. 2013.

[9] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Proc. 16th Working Conf. Reverse Eng.*, Oct. 2009, pp. 31–35.

[10] B. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, Boston, MA, 1999.

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Computing Research Repository*, vol. abs/1301.3781, pp. 1–12, June 2013.

[12] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[13] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[14] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008.

[15] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.